Slides for the Presentation at The John H. Glenn Research Center at Lewis Field November 2004 Technical Narrative

Dr. William Henry Jones

November 2, 2004

The following pages provide a technical narrative that more fully elaborates upon the associated slide set.

1 Slide pst_vgrf_0130 – Project Integration Architecture

This presentation will give a brief overview of the Project Integration Architecture (PIA) effort being conducted at the National Aeronautics and Space Administration's John H. Glenn Research at Lewis Field in Cleveland, Ohio.

Project management responsibilities for PIA have been exercised by various individuals over the passing years. The current designated project manager for PIA is Ms. Theresa Benyo; however, Ms. Benyo has been detailed over much of the past year as the project lead for the Glenn-lead *Inventing Flight* celebration of the centenial of flight. During Ms. Benyo's absense the Chief of the Engine Systems Technology Branch, Mr. Richard Blech, has been performing the project management role for PIA.

Dr. William Jones is, and has been since its inseption, the Technical Lead for the PIA effort. A precursor effort, the Integrated CFD and Experiments (ICE) effort, was lead by Mr. Dale Arpasi. Some small number of other personnel have assisted in the PIA technical effort; however, without exception all have moved on to better-compensated positions both at the Glenn Research Center and elsewhere.

Finally, active efforts at commercialization and actual application of PIA technology have recently been made by Battelle Memorial Research Institute (through a commercialization activities contract with NASA-Glenn), Entara Technologies Group, LLC, (through a Software Use Agreement and a pending Space Act Agreement), and C. Harnett Teska (through a Software Use Agreement). C. Harnet Teska was recently admitted to the Center for Advanced Technology and Innovation in Racine, Wisconsin, as one of only six incubator projects now supported by that government/private/university consortium.

2 Slide pst_vgrf_0112 – PIA: The Oversimplified Nutshell

PIA is an effort of considerable scope; thus, the reduction of its intent to a single, short statement is likely to lead to more mis-understanding than clarification. Nevertheless, some such nutshell explanation seems generally inevitable. Thus,

> Project Integration Architecture (PIA) is a distributed, object-oriented, architectural framework that provides (in a machine-intelligible manner) for the generation, organization, publication, integration, and consumption of all information involved in any process.

Putting this another way, the intent of PIA is to organize all of the information generated and stored on computers. Furthermore, the intent of PIA is to make this information intelligible not simply to people browsing away through some portal, but to other computers so that applications can be meaningfully linked together to form super-applications, applications searching for some meaningful starting point or touch-stone can, themselves browse through such inforation seeking what they need, and so on. Finally, it is the intent of PIA to generate auditable records of what went on, where information came from, and the like so that, when all the dust settles, one can actually know by what process an answer was developed.

Making PIA an even more interesting project to understand, there is very little restriction on the sort of information and process that PIA can accommodate. The basic PIA model is that information may or may not be put into the computer, something may or may not be done to that information by the computer to add to the value of that information, and information may or may not come out of the computer. There is no further restriction to the PIA model: there is no specification that PIA deals with analysis or design application, no specification that databases of information are served, no restriction to aerospace applications or automotive applications or pharmacological applications, nothing more. Simply data might go in, something might happen to it, and something might come out.

To conclude the confusion, one must understand that, in and of itself, PIA does nothing. It is a framework, a methodology, but not an actual application itself. It is somewhat like the Dewey Decimal System for indexing libraries:

just sitting there, the Dewey Decimal System is without value; it is only when a librarian actually uses it to index the books of a library that the Dewey Decimal System take on utility. If it just so with PIA: it is only when developers wrap their information sources and sinks in PIA compliant technology that PIA does something useful.

3 Slide pst_vgrf_0093 – Key Object-Oriented Technologies Exploited by PIA

In order to accommodate the extremely laid-back expectations of the PIA information/process model, two key object-oriented technologies have been exploited to an extent perhaps beyond any previous effort: the technology of self-revelation and the technology of semantic infusion through class derivation.

The first technology, that of self-revelation, is the ability to inquire of an object as to its nature; that is, for an object to reveal its nature to its consumer rather than require its consumer to fore-know the nature of the object.

The natural analog of this technology of self-revelation is experienced by a person every time he or she meets someone new. Other than the very general expectation that the new person is a human being, fore-knowledge of new acquaintances is generally very small. It is through self-revelation that we begin to know another. We inquire, we ask questions: what do you do for a living, are you married, do you own a home, so on and so forth. The answers guide our further interactions; a person who turns our to be a medical doctor guides us into our current need to inquire about some pending medical condition and perhaps obtain an appropriate referral.

Self-revelation allows programs to be built in much the same way. Instead of being programmed with a rigid expectation that needed information must be found in exactly a particular place and in exactly a particular way, programs may be built with flexibile scenarios willing to respond to information in the way and place it is encountered. Clearly, programming for this added flexibility is an added burden, but it is also an added advantage. Programs do not break because another program providing a needed nugget of information is removed and a third providing that information is swapped in as a substitute, nor need they break because in one situation B follows A while in another A follows B.

Self-revelation is considered in two slightly different forms: a self-revelation of kind and a self-revelation of content. Self-revelation of kind addresses just that: of what kind is a particular object. This question and answer sets the expectations about the nature and content of the revealing object. It is as with a person answering "I am a medical doctor": it sets your expectations as to the kind of skills and abilities that you will find in the person.

The complimentary aspect of this self-revelation is then the self-revelation of content: to what extent are expectations, in fact, fulfilled. You are a thorasic surgeon: how many years of experience do you have, how many operations have you performed? If this doctor has done 1,000 heart valves, perhaps this is the doctor for you; if he just entered residency, probably not. Similarly, a code looking for a flow-field solution to compare its results to can inquire as to the validity of a found solution, can compare the geometries of the two solutions, and the like to determine whether or not this is an appropriate point of comparison.

The other technology necessary for self-revelation to be meaningful is that of semantic infusion through class derivation. This is the technology by which the nature of an object is refined by deriving it from a more general object. As the successive layers of derivation are built up, the nature of the object becomes more and more specific, and as a consequence the self-revelation of kind becomes more useful and specific. Continuing the analogy with a person, you find out first that the person is a working professional, that the person is a doctor, then a medical doctor, then a cardiologist, and so on. As each new layer is added, what you may then infer becomes more specific and more valuable in determining whether this person is the person that fits your needs.

4 Slide pst_vgrf_0132 – Semantic Infusion Through Class Derivation

This graphic provides a specific example of the technology of semantic infusion through class/interface derivation by illustrating the heritage of a specific parameter interface developed for the the CORBA-served version of PIA. This interface starts from the patriarch of the CORBA interface system, the **CORBA::Object** interface, and is progressively derived and defined until, at last, it is specifically a parameter of the LAPIN application wrapper encapsulating a 1-dimensional grid of corrected massflow results.

Each layer of derivation contributes to the whole as follows.

1. **CORBA::Object**: Instances of ll interfaces served by CORBA must be derivatives of this foundational interface.

2. **GObject**: This is the patriarch of the entire PIA interface hierarchy. As such, it does very little except declare that it is an instance that is willing to reveal its nature (self-revelation) in various ways.

3. **GObjSta**: This layer simply adds some boolean characteristics to an instance; that is, it adds things that may be either true or false about a particular instance.

4. **GObjLck**: This layer adds the ability to lock access to each particular instance, thus providing protection against possible information corruption through uncontrolled, concurrent access by multiple accessors. Traditional multi-reader, single-writer controls are provided through **Release**, **Reference**, **Read**, **Write**, **Execute**, and **Delete** lock levels.

5. **GObjDgn**: This layer adds to the instance the ability to participate in a directed graph. A directed graph is a very general and, often, very useful structural form that allows relationships between instances to be recorded.

6. **GacBObj**: At this point, the instance is declared to be a functioning part of an application, although just exactly which part and which application (if any) is still unspecified. In PIA the most useful thing that being a part of an application brings with it is the ability to be

described by a very flexible descriptive system, that being a form of self-revelation of content. Another thing that comes with being a part of an application is the ability to search upwards through the defined structure of an application to find some particular layer; for example, a parameter can search upward to find the problem configuration of which it is a member.

Through an object-oriented slight of hand, the instance locking system introduced by the **GObjLck** interface layer is expanded into a full per-user/per-instance access control system. The concept of a "user" is introduced and the protocols extended to determine not only whether or not a particular user can gain the requested access now, but whether or not that access can be granted at all at any time. The concepts of ownership and other privileges are also introduced in the form of new, distinct lock levels.

A number of other useful descriptive elements can be applied to any derivative of the **GacBObj** interface. For example, digital signatures can be attached so as to verify the state of an instance at a point in time. This is expected to be useful in the commercial world for the protection of intellectual property rights.

7. **GacPara**: This layer declares that the part of the application is, in fact, a parameter. A parameter is defined as being the principal information-bearing kind of instance: the input texts, the output numbers, what have you. Parameters are held in problem configurations (groups of parameters defining the specific state of a more generalized problem being solved) and parameters can participate in dependency graphs so that a change in one parameter can be correctly reflected in parameters dependent upon that parameter.

8. **GacParaArr**: This layer declares the parameter to be structured in the manner of a 1-dimensional array (also sometimes regarded as being a vector in n-dimensional space); that is, whatever it is, there is a set of it from which any particular member may be uniquely identified by a single index value.

9. **GacParaArrDoub**: This layer declares that the array parameter has a specific type: a double-precision floating point (real) number.

10. **GacParaDimArr**: This layer declares that the floating-point numbers organized by the array are dimensional in their nature; that is, the numbers represent measurements in some unit system. That unit system is encapsulated in an associated description of the instance using the general mechanism defined back at the **GacBObj** derivational layer. This dimensional interface layer turns off the ability to deal with these numbers in any but a dimensionally-aware way.

11. **GacParaDimArrMfl**: This layer declares that the dimensionality introduced in the previous **GacParaDimArr** layer is the dimensionality of mass flow; that is, it is a measurement in terms of mass per unit time.

12. **GacParaGasArrMfl**: This layer declares that the encapsulated massflow numbers are, in fact, the massflow of a gas. This means to any consumer of the numbers provided that they refer to a gaseous fluid such as air, not a liquid and certainly not a solid.

13. **GacParaGasArrMflCr**: This layer further defines the encapsulated gas massflow numbers to be corrected values. (Corrected massflow applies a correction factor relating ambient total pressure and temperature conditions to sea-level static conditions.)

14. **GacParaGas1DGridMflCr**: This layer refines the semantic meaning to indicate that the encapsulated numbers represent a 1-dimensional grid of corrected massflow values, presumably obtained through analysis or, possibly, experimentation. An expectation is now added that a related parameter will encapsulate the corresponding axial positions of the grid points for the corrected massflow numbers.

15. **LapPar1DGridMflCr**: Finally, this derivational layer indicates that this parameter is actually produced by the LAPIN analysis code. General consumers of information need not know or care what LAPIN actually is, though this could be discovered through analysis of the containing **LapAppl** application instance; however, by providing a LAPIN-specific parameter derivation, the LAPIN wrapper, itself, is able to recognize its own parameters in case it has encapsulated application-specific capabilities in them.

As may be seen, this process of derivation has taken us from an amorphous blob of an instance to something very specific and very clear, something that

a computational fluid dynamics code trying to solve the flow through an inlet could find, recognize for what it is, and put to use in establishing some initial condition of the flow.

Another aspect of PIA's implementation of self-revelation is illustrated by this diagram. PIA's self-revelation supports a concept of depth: not only can it be discovered what an instance is on its surface (here, a LAPIN-generated 1-dimensional grid of corrected massflow results), but the instance's layers can be peeled away and its nature all the way to its core discovered. The process of doing this is called **ecdysiastical analysis** (from the Greek *ekdysis*, from *ekdyein*, to get out of, strip off). By doing such an analysis, some codes can deal with whole classes of information at the level suitable to the situation.

As a practical example of ecdysiastical analysis, consider a Graphical User Interface (GUI) that simply wishes to display numerical values. The code to do this can cast aside all the specifics of closely defined, semantically-infused parameter interfacees and deal with a **LapPar1DGridMflCr** array, as well as a whole host of other such arrays, as simply being a **GacParaDimArr** array of dimensional values. The GUI can request the values in the unit system currently selected by the user and display the values in a scrolling edit box, or as an X-Y plot of value versus index, or whatever. The GUI doesn't really care if the values are a gas corrected massflow or anything else: they can be usefully displayed for the user simply as an array of values.

5 Slide pst_vgrf_0055 – PIA Application Architecdtural Wall Concept

Conceptually, PIA builds a consistent architectural wall between consumers of applications and information and the actual application and information resources. This wall is shown in this diagram as the column of blue blocks, each conceptually connecting on its left in an orderly top-to-bottom, right-to-left, plug-and-play manner. The job of each blue block is to adapt from the world of order on the left to the world of individual confusion and chaos on the right. In this world on the right are the different sorts of information and application resources: databases of experimental information, archives of geometry information, application codes capable of turning inputs into outputs, what have you. It is even possible for there to be no oddly-colored block at all; a PIA wrapper can be the entirety of an application in and of itself.

The advantages of the world of order shown on the left are relatively obvious.

1. Common tools can be used to access all sorts of information. For example, a single Graphical User Interface (GUI), perhaps built with features and abilities appropriate to the discipline of its user, can provide access to all manner of applications without ever having been specifically informed of any such application.

2. Search engines and browsers can be developed that roam over the whole of offered information without ever needing fore-knowledge of the kinds of information that will be encountered. (Compare this to current database access tools that must be built with an explicit knowledge of the record formats that will be encountered.)

3. Most importantly, applications themselves can search out into this world of order to interact with other applications without the need of human direction. For example, a computational fluid dynamics code charged with solving a posed flow field problem can, on its own (programmed) initiative, search for a similar solution from whatever databases of experimental results it can find.

Another point illustrated by the bottom two wrapper blocks of this diagram is that a single wrapper need not be the only access path to an applica-

tion or other information resource. For example, "journeyman" and "master" wrappers can be devised to the same application code: the first providing heuristics and other assistances to allow the less-knowledgable user useful, but safe, access to the application while the second allows the assured master of the situation to turn every control as he sees fit to test the limits of validity and performance.

6 Slide pst_vgrf_0094 – PIA Self-Revealing Application Architecture

6.1 The Basic Structure

This diagram gives a more technically accurate depiction of the "application" architecture defined by PIA. An application is represented by a single, coordinating object, labeled **PacAppl** in this picture, from which three principal structures emanate.

1. **Parameter Configuration Tree**: The actual parameters of an application are held in a parameter configuration tree (the middle structure emanating from the **PacAppl** object and proceeding toward the lower left corner of the diagram) which organizes them into distinct configurations of the problem being studied. The configurations track the path of investigation: typically, a number of sibling configurations are studied, a "best" one or two are selected, and investigation proceeds downward from those selected points.

2. **Parameter Identification Tree**: The identification and structure of the parameters of an application is revealed by a parameter identification tree (the right-most structure emanating from the **PacAppl** object).

3. **Operation Map**: The operations that a particular application can perform are revealed by an operations map (the left-most structure emanating from the **PacAppl** object).

Each of these structures is discussed at greater length in the following sections.

6.1.1 The Parameter Configuration Tree

As mentioned in the enumeration above, the primary information – the data, parameters, and the like – of an application is held in an *n-ary* tree of parameter configurations. The blocks representing these are labeled **PacCfg** in the diagram. Each of these configurations represent a distinct point of investigation in the encapsulated application. For example, in an experimental application, each configuration might represent a distinct setting of the

experiment at which data was acquired. In the analysis of a design, each configuration might represent a particular point in the design space that was evaluated and compared against other configurations of the design.

Each configuration then contains in itself a map of organized parameter objects sorted by the fully-qualified name developed for each parameter identification. (The development of the fully-qualified name is discussed in the next section.) These parameters are represented in the diagram by the block or blocks labeled **Par: x/y/z**, where the **x/y/z** portion represents the fully-qualified name associated with the parameter.

The general intent of the parameter configuration tree is to track the progress of problem investigation. Generally, it is expected that, at any particular juncture, a number of possibilities will be studied and that these various possibilities will be well represented by a set of siblings in the parameter configuration tree. The expectation continues by suggesting that of these sibling configurations some few, perhaps as few as one, will be selected and represent the best choice for advancing the investigation. The parameter configuration tree will then represent further investigation as descendent (or offspring) nodes from the selected siblings, again giving rise to a new set of siblings. As various alternatives prove to be less than fully competitive, their branches of the overall parameter configuration tree will simply be abandoned while more competitive selections will continue further growth until, ultimately, some final, best configuration is found.

Because this process of investigation is expected, usually, to involve only small changes from configuration to configuration, the parameter configuration tree introduces the concept of parameter inheritance: a needed parameter missing in a particular configuration is considered to be inherited from the most proximate ancestor of that configuration actually containing that parameter. In this way, investigations involving merely the tiggling of a few key parameters can avoid the burden of replicating the entire parameter set from configuration to configuration.

Naturally, situations exist in which the parameter configuration inheritance protocol is inapproprate. Consider, for example, an experimental data application in which each configuration is actuallly a complete data sample: in such a situation it would be inappropriate to inherit experimental readings from other samples when the readings are missing due to instrumentation that has failed during the intervening period. To accommodate such situations, the parameter inheritance protocol can be turned off on a case by case basis. The implementation of this option is a part of the parameter

identification mechanism that is to be discussed next.

6.1.2 The Parameter Identification Tree

The parameter identification tree actually arose from the concepts of the parameter configuration tree, specifically the parameter inheritance protocol in which a parameter missing from a particular parameter configuration node can be inherited from the most proximate ancestral configuration actually containing that parameter.

While not always the case, parameters are often structural in nature, existing as a coordinated unit rather than simply as isolated values. For example, some computational fluid dynamics codes express their flow fields as multiple blocks having the same structural form, but different specifics suited to the nature of the flow in the region the block covers; the same parameters repeat from block to block, but contain different values. Usually, these structuralizations are represented as a literal pattern of data: some key item introduces a new structure of data and then the pattern is followed again to identify the various parts.

Because PIA's parameter configuration tree would like only to create the pieces of data the distinguish one configuration from its ancestral line, this literal structuralization of data produces a problem: how does one identify the structural unit a particular item belongs to when all of the structures are not necessarily present in the configuration? The parameter configuration contains no key to introduced the beginning of a structure and the patterned elements of each structure do not exist unless they represent a material difference from an ancestral configuration.

The parameter identification tree was introduced by PIA to deal with this difficulty. That structuralization of parameters is encoded into the structure of the parameter identification tree. Each node of the parameter identification tree is given a name, those names being unique among siblings in the tree. Each terminal node of that tree then identifies its corresponding parameter by concatenating the names from terminal node to patriarch to produce a fully-qualified name by which the parameter is known within any parameter configuration. By so doing, the revealed structure of data is flattened into a single text name and the need to replicate data within a configuration to preserve structure is eliminated.

As an example of this structuralization, consider again the multi-block computational fluid dynamics code wrapper. One approach to parameter

identification would be to repeat a flow block identication structure once for each flow block actually involved in the solution of the problem. The name of the head of each flow block identification subgraph would, of course, have to be unique while the names of the various parameters within each subgraph could repeat. Each developed fully-qualified parameter identification name would be unique at the flow block identification level. If the course of investigation involved only the tiggling of the parameters of a few flow blocks out of many, those parameters would be clearly identified in their configurations as pertaining to the flow blocks of interest by their structuralization-flattening names.

The parameter identification tree is also the mechanism by which the parameter inheritance protocol of the parameter configuration tree can be turned off. In simple cases, this may simply be the disabling of the support mechanism in the particular identification node of the tree. Greater sophistication is, of course, possible. A customized, derivative identification node can be developed with knowledge of particular situations when such needs arise.

The identification mechanism implements another feature: the ability to report whether a particular parameter is, in fact, visible. An invisibile parameter, even though it might actually exist in a configuration (or that configuration's ancestral line), is reported as not existing. This ability may be used to disable the parameter inheritance protocol when that is appropriate.

To understand the visibility feature, consider a situation in which an analysis application has an optional model with a number of parameters that specify the operation of that model and a single parameter that turns that model on and off. The identification nodes for the specification parameters may be developed with knowledge of the **on/off** selection parameter so that they will report the specification parameters as being invisible when the **off** selection is the visible selection. In that way, the specification parameters will appear not to exist, even though they might exist in (or be inherited by) a particular configuration that has (or inherits) the **off** selection parameter.

6.1.3 The Operation Map

The final element of the application architecture displayed in the diagram is the operation map, illustrated by the boxes labeled **PacOp** in the diagram. This is simply a map (sorted by name) of encapsulated operations that the application is willing to do. Typically, one such operation converts input

parameters into output parameters by running the encapsulated application; however, such an operation is not a requirement and is, indeed, very likely to be absent in applications that simply serve archives of information. There may be other operations that such applications may offer. For example, an experimental database application might have the ability to extend its supply of offered information by checking with its wrapped experimental facility.

Operations, in a manner similar to parameter identification visibility, have the ability to report whether the encapsulated activity can currently operate. For example, a run operation might check to see if the necessary input parameters exist before attempting the operation. While the encapsulated operation, itself, is expected to check and honor pre-requisites (rather than simply accepting the invokers assurances), the ability to inquire first is provided to allow interactive consumers such as Graphical User Interfaces (GUIs) to gray-out or otherwise indicate that the possibility is not currently available.

6.2 Operation in Context

The diagram shows some graceful, sweeping curves from both the terminal nodes of the parameter identification tree and from the nodes of the operations map. As may have been anticipated, both of these mechanisms operate in the context of an identified node of the parameter configuration tree. For example, nodes of the operation map determine whether or not they are enabled for operation by examining the parameter content of the parameter configuration node provided to them; a run command disables itself if it does not see the necessary input parameters in the identified configuration. Similarly, the parameter identification nodes make their visibility and inheritance judgements based upon the information in the configuration identified to them.

6.3 The Ecdysiastical Sorting Structure

A fourth structure is actually included in the PIA application architecture; however, due to its complication the structure is not represented on the diagram. This structure is a complete, ecdysiastical sorting of all the information-bearing objects existing within the application. This object set is generally considered to begin with all of the parameter objects existing in all the configurations of the application; however, the set is not limited to

only parameter objects. Other objects such as selected kinds of description objects may also be included in this set.

The ecdysiastical sorting of a set of objects is a comprehensive, layer-by-layer sorting of those objects, rather than just a sorting by the surface type of an object. Thus, an object with 15 layers of derivation from the patriarchial **PObject** class is included in 15 separate sortings corresponding to each of those derivational layers. By doing this, objects can be identified at the derivational level at which it is intended to deal with them, without regard to whether or not they are exactly of that derivational level, or of some level derived from that level.

The inclusion of the ecdysiastical sorting in the application architecture is thought to be important since applications may choose to make all of their parameters customizations beyond the well-known. For example, an computational fluid mechanics code named **Xyz** may choose to further derive a far-field, upstream Mach number parameter into an **Xyz**, far-field, upstream Mach number parameter. A sorting of information by its encapsulating object's surface type would be of no use to an outside consumer of information since it would have no knowledge of anything specific to **Xyz**, but an ecdysiastical sorting would allow that outside consumer to go straight to far-field, upstream Mach number parameters without concern that in this case they were **Xyz**, far-field, upstream Mach number parameters.

7 Slide pst_vgrf_0095 – Integrated Application Graphs

Having wrapped applications in PIA-compliant wrappers enables the next step, one of the principal goals of the PIA effort: the integration of many applications into a multi-fidelity, multi-disciplinary, cooperative whole. This diagram illustrates this with an (imagined) interconnection of applications used to analyze the performance of a proposed Rocket-Based, Combined-Cylce (RBCC) engine intended to propel a single-stage to orbit vehicle.

The flow of information starts with a geometric definition of the engine. That definition was, in fact, held in a commercial Computer Aided Design product. (In an actual PIA prototype demonstration effort, this geometric information was successfully accessed through a PIA-compliant wrapper that was built upon the Computational Analysis PRogramming Interface (CAPRI), a vendor-neutral geometric Application Programming Interface (API) technology developed by Dr. Robert Haimes of the Massachusetts Institute of Technology under a grant from the Computer and Interdisciplinary Studies Office of the Glenn Research Center.)

From the geometric definition of the engine, information would then flow to APAS, an airloads panel code, GASP, a more comprehensive computational fluid dynamics analysis code, and NASTRAN, a well-known, commercial finite element analysis code. As depicted by the diagram, information generated by these components would flow on to other components until, ultimately, some sort of answer would come out of the bottom indicating the merit of the proposed engine design.

This diagram then adds an imagined great recirculation to the top in which a new configuration of the engine is proposed, a new geometry entered as a new PIA parameter configuration, and the process restarted.

A key element in making such an integration work is the self-revelation and semantic infusion technology PIA is built on. In typical integration technologies, it must be specifically explained by the integrator just exactly where each integrated application is to find its input, how it must transform what it finds into what it needs, and where it should put its products so others can find it. With PIA self-revelation technology, an application is simply connected to another and, upon an appropriate nudge in the side, looks for itself to see what it can find and, based on what it finds, decides for itself what use it can make of that information. The coding for such an effort

is, of course, difficult, but the payoff is significant: a wrapper is not coded for connection only to some other specific wrapper, but instead is coded simply to look up the line and see if it can find the information it needs.

In actuality, this integrated, comprehensive analysis has not been done because of the effort involved in developing all of the PIA wrappers to all of these codes. What has been done, though, is a much simpler effort in which the actual engine geometry was extracted through a PIA wrapper and the relevant information transported automatically to a PIA-wrapped flow stability analysis code not shown on this diagram. This effort demonstrated the validity of the concepts and assured that, should effort extend to such a more sophisticated analysis, those basic ideas of information flow would in fact work.

8 Slide pst_vgrf_0133 – Autonomous Solution Systems

8.1 The Need and Basic Idea

The ability to flexibly integrate applications into comprehensive, multi-fidelity, multi-disciplinary analyses of large, complex systems opens up a new area of opportunity, but also a new area of difficulty. Experience with current, commercial integration technologies suggests that the raw number of details involved in an integration begins to overwhelm the human being as the number of elements grows beyond the general range of 15 to 20 applications. Even with the more flexibile and adaptive integration technologies demonstrated by PIA prototype efforts, this number may not be significantly greater because, while relieved of the actual detail effort, the human integrator must still keep in mind the kinds of information that must be generated and the general causal flow of that information. This number of 20, 30, or 40 applications managable through even the enhanced technologies of PIA must then be compared with industry goals of integrations on the order of 1,000 applications or more. Indeed, Boeing Aircraft Company estimates that it has a total of some 10,000 active engineering applications which it would like to see operable in a cohesive, integrated manner.

It is thought that the PIA technology of self-revelation again offers a potential solution to this problem. As objects in general have been devised to reveal their kind and their characteristics, it is entirely natural to devise application objects that are willing to reveal the products they produce (in terms of the kinds of parameter objects generated as the output of the operation of the application) and the inputs needed to produce those products. Using this information, it is then proposed that an algorithm can be devised to assemble application graphs that solve a problem posed in terms of the desire to attain an optimal value of a particular result. With such an algorithm, the application integration selection process would be converted from an essentially manual process performed by a person (with the help of computer-based tools) to an automatic process performed by a machine.

8.2 The Essense of the Algorithm: Program Linkage Editing

The essense of this algorithm has already been in use on a daily basis since nearly the dawn of electronic computing. It is the program linkage editor, often known simply as the "linker". The linker is usually given some initial, incomlete chunk of programming, usually a program named *main*. This program has, in one way or another, two tables: a table of entry point symbols which it defines and a table of entry point symbols for which it needs definitions to be made complete. The linker drops this program chunk into the building program image, notes the entry points that are now defined somewhere within that chunk, and adds the entry points for which definitions are needed to the linker's own internal table of things it is still looking for. The linker then proceeds on to other bits and pieces of programming supplied to it: other code modules, specifically-specified libraries, standard libraries, and the like. As the linker browses through each of these programming sources, it keeps in mind the list of things it needs. When it finds a chunk that satisifies a need, it drops that chunk into the program image, moves the satisfied entry points from the needed table to the defined table, and adds to its own table of needs anything new that the added chunk needs. This process continues until the needed definition table is empty. If the linker comes to the end of its browsing and still has things that it needs definitions for, it then knows that it is impossible to create the desired program and prints an error message.

The automatic generation of an application integration graph to solve a posed problem is seen as being this same fundamental linkage editing algorithm, except that now the symbols are not entry points in program code, but kinds of parameter objects. For example, let us pose a problem as being a desire for the best cost per pound to low earth orbit. Obviously, we will need a **CostPerPoundToLEO** parameter as our final result, so we tell our algorithm to put that in its needed parameter table. The algorithm is then told to solve that problem. It looks at its table and realizes that it will need an application that produces **CostPerPoundToLEO** as its output, so it searches the PIA environment for an application claiming (through self-revelation) to produce that result. Finding such an application, the algorithm

1. Adds the application to the building application graph,

2. Takes **CostPerPoundToLEO** out of the needed table and puts it in

the produced table,

3. Asks the application what it needs to make that particular result, and

4. Puts those parameter needs into the needed table (assuming, of course, that those inputs are not already available).

This process then continues on, looking for applications that produce the kinds of information that still reside in the needed parameter table.

The application graph assembly process ends in a manner slightly different from the linkage editor. The linkage editor stops when its needed entry point table is empty: simply, it has found every thing it needs and an explicit answer to its problem has been found. That should not be the case in application graph assembly because it leads to the conclusion that there is only one answer, not good and bad answers. Instead, the application graph assembly algorithm should stop when it finds that all its needed inputs are guessable on a random basis. The technical term for this is that its needed inputs form an independent design vector. This characteristic is, again, something that can be identified through the technology of self-revelation: as each needed input parameter is identified, it is asked whether it is a randomly-guessable input; if it is, the algorithm would so note and not try to find an application to satisfy that particular need.

The significance of randomly-guessable inputs (the independent design vector) is that, given any particular set of such random numbers, some design is specified and may be analyzed. It may or may not be a good design, but it is at least a design. Having this, the problem is now reduced to one of figuring out which set of numbers produces a good, and even a best, design. This task comes under the broad term of "optimization", which has already been, and continues to be, extensively studied.

8.3 After Assembly; The Optimization Process

The topic of optimization of a design given a method of solution is really an issue subsequent to the autonomous formulation of the solution graph. Nevertheless, a couple things might be noted at this junction. Because real, complex systems are likely to have large design vectors, the optimization of a design is likely to be very challenging. The Langely Research Center, among other organizations, has been working on technologies to partition such a large problem into smaller, more managable units that may be optimized

quasi-independently in a manner still in concert with the whole. Such technology will probably be vital. Beyond this, more common technologies will probably divide the overall optimization process into distinct phases along the following lines.

1. A statistical characterization phase may be able to sort out the independent parameters that have significant effects upon the design. This will allow less important parameters to be ignored, or at least neglected until phases of final refinement begin.

2. A genetic manipulation phase may be of use in identifying the region of a global optimum with relative speed, thus avoiding a lengthy optimization into what is only a local optimum.

3. A true optimization phase would then refine the selected design. Less significant independent parameters might be re-involved in the design process during later stages of this process.

4. A final "six-sigma" assessment of the design might further refine the design, possibly backing away slightly from a truly optimal result to obtain a more-reliably manufacturable design.

There are, of course, a great many details to real optimization processes. One in particular is that, in general, design spaces are not unconstrained. Many parameters will have constraints: usable and ultimate strengths of materials, minimum spacing of fastners, maximum response rates for controls, all manner of things. Again, it is expected that the technology of self-revelation will allow these and other issues to be dealt with in a flexible, adaptive manner. The optimizer will have to ask each of its variables and each of its applications about those things and arrange itself so as to meet those specified requirements

9 Slide pst_vgrf_0086 – Continuing the Autonomous Assembly of an Application Graph

Under the assumption of a sufficiently rich PIA environment, an application producing the previously needed result of *cost per pound* to low earth orbit is found and becomes the basis of the application graph to be assembled.

The algorithm then inquires of that found application to determine what parameter it needs as input to produce the *cost per pound* output and finds that, in this simple example, the application needs *cost* and *pounds*. The algorithm moves the *cost per pound* parameter from the needed list to the found list and places *cost* and *pounds* on the needed list. The search for applications then continues on.

10 Slide pst_vgrf_0087 – Further Recursion of the Autonomous Assembly Algorithm

Since the needed parameter list has not been reduced solely to those parameter guessable on a a random basis (that is, to an independent design vector) the autonomos application graph assembly algorithm searches on for more applications producing the currently needed parameters, *cost* and *pounds*. Again, it finds such applications and adds them to the building application graph. The now found parameters *cost* and *pounds* are moved to the found parameter list and the parameters that the newly-found applications need to operate, *fraction*, *gross*, *per use*, and *fixed* are put on the needed parameter list. Operations then continue.

11 Slide pst_vgrf_0088 – Reduction to Applications Requiring Only Random Inputs

The application graph building process continues until the needed parameter list kept by the alorithm consists only of inputs that can be guessed on a random basis. This set would then constitute an independent design vector suitable for manipulation by an optimization process.

12 Slide pst_vgrf_0110 – A Rocket Motor Design Application with Random Inputs

As a very simple illustration of an application requiring inputs that can be randomly guessed, consider the geometric design of a simple rocket motor. The essense of a rocket motor is captured in three simple numbers: the cross sectional areas of the combustion chamber, the throat, and the skirt. Any three random numbers will give you a basic rocket motor design. Unfortunately, most sets of three random numbers will give you a very bad rocket motor design, but they will, nevertheless, give you a design. It is up to the optimization phase to figure out which three random numbers give a best rocket motor design for the given situation.

One can improve this design application by switching the usage of the numbers. Instead of simply accepting them as the direct specification of the three areas, a more sophisticated application might accept the first number as specifying the cross sectional area of the throat (which is often dealt with as being the key number in rocket motors) while the two succeeding numbers would be taken to be the ratio of the two other cross sectional areas, the combustor and the skirt, to that throat area. If the application were then to put a stipulation on those two succeeding numbers that each must be greater than unity, then the design space would be at least limited to traditional convergent/divergent rocket motor designs.

13 Slide pst_vgrf_0106 – Applicaiton of Solution Initialization and Inprovement Technology

Once an application graph has been built by the autonomous solution system algorithm (or by manual methods, as the case may be), some other entity must pick an initial design (in the form of specific values inserted into the developed independent design vector), determine the merit of that design, and then vary those values to improve that design. This entity, and activity, is shown as the block to the right in the diagram indicating the supposed phases of such an activity:

1. Statistical characterization,
2. Genetic manipulation,
3. Optimization, and
4. Design for Six Sigma.

This overall design improvement activity is considered to be outside the actual province of PIA development: PIA focuses strictly on the foundational core of information and application representation and integration technology. A design improvement module, like a graphical user interface, a browser, or a search engine, is considered to be a consumer of PIA resources, not a PIA resource in and of itself.

14 Slide pst_vgrf_0107 – Use of Relevant Experimental (or Other) Information

While the main use of PIA technology thoughtout the discussion of autonomous solution systems has been for the integration of a developed application graph, this does not preclude wrappers within that graph from reaching out to other PIA resouces not directly participating in the solution process.

There are some areas of engineering analysis, computational fluid mechanics being one well known at the Glenn Research Center, that simply do not start and operate well from an arbitrary starting point. These analyses, generally, need some reasonable beginning solution which they can improve to a significantly more accurate condition. In some cases, it may be that some pre-cursor application in the application graph will provide that "rough-in" analysis (indeed, that was one of the original multi-fidelity analysis integration goals of PIA), but that may not always be the case.

Fortunately, the ability of a PIA application wrapper to use PIA resources just as any other consumer of PIA-represented information would provides the opportunity for a remedy. A wrapper able to recognize an insufficient starting condition would be entirely within the PIA conceptual bounds if it were programmed to browse through other PIA resouces in search of some appropriate starting point. Taking up the computational fluid mechanics example again, such a wrapper could browse through other PIA-wrapped archives of experimental and analytical flow field results searching for a result to a similar problem. Since PIA makes the association of all sorts of information possible, it is expected that the geometry of found flow field results could be identified and that the seeking wrapper could compare the geometries of found results to that of its own problem to further inform the decision process as to whether or not a particular found result represented, in whatever sense, a good starting point for the problem at hand. Such a search is the situation depicted in the diagram.

PIA provides further facilities that extend even this situation. It is possible to devise application wrappers that are aware of a supporting help-desk facility, presumably staffed by expert humans, for their wrapped application. In appropriate situations, a wrapper could communicate with that help-desk to indicate its concerns and await direction. The help-desk might run experiments or independent analyses to provide a starting point and direct

the wrapper to them when they become available. As an alternative, the help-desk might examine the overall situation and advise the wrapper that the proposed effort is outside the bounds of reason; the wrapper's response would probably be to make a notation to that effect and inform the rest of the application graph that the particular configuration of the problem is untenable. The possibilities for such a mechanism are limitless.

15 Slide pst_vgrf_0098 – Autonomous Solution System Benefits

The advantages of autonomous solution systems are many. First and foremost is that integration beyond the limits of human fuddling is enabled. The tendency of the human, or even the team of humans, to get confused as too many facts get tossed into the air at once is replaced by the mindless, nearly-inerrant plodding of the machine. Linkage editors today flawlessly assemble programs with tens-of-millions of entry points; integrating a mere 10,000 applications into a particular solution without muffing a single detail ought to be like falling off a log by comparison.

To further understand the benefit of this automated extension, consider a human being attempting to integrate 400 applications, each with an average of 50 connections to make to other applications of the integration. This would require the making of 20,000 connections. If a person with the proverbial 99.99% accuracy attempted this feat, there would be an 86% chance that at least one of those connections would be wrong. This is far too great a probability to be tolerated in a great many businesses from space flight to the design of super-tankers.

Akin to this first advantage is that the same problem may be re-solved as easily when new resources come available. With the advance of knowledge, new resources of design and analysis are certain to become available. Revising a massive human effort of integration in order to incorporate a few new wrinkles might give managers some pause; however, simply double clicking an icon to see how the solution shakes out today with the new resources in the system would give few any concern.

Another possibility that arises is that applications might eventually be validated for the quality of their analysis; a number that indicates how good their results are. Starting with some arbitrary value for the random inputs feed into the solution, these accreditation values could be applied to assess the validity of the final output. Furthermore, automated analysis could be done of the progress of validity through the course of the solution graph. Areas of weak ability, in which the validity of the progressing solution did not increase steadily, could be identified. Indeed, some applications might assess their validity based upon the actual values of the current problem configuration (for example, a computational fluid dynamics code might correlate its validity to its achieved convergence) and would be able to contribute to the

identification of less valid areas of the design space.

This concept of result validity might then be used to consider alternative solution strategies. An over-rich analysis environment has the potential of solving a given problem more than one way. Two approaches to this dilemma suggest themselves. One approach is to use self-revelation to extract validity estimates at the time of application graph assembly and select the solution approach that seems to offer the best results. The alternative approach is simply to assemble the multiple solution methods, probably as independent solutions, exercise them all, and see after the fact which gave the best solution.

Finally, another key benefit of integration technology in general which is made more complete by autonomous solution system technology is that discipline expert's team participation time may be significantly reduced. Instead of attending weekly team meetings to go over the current results and decide what cases will need to be analyzed next, and then spending the rest of the week turning the crank on those cases, the discipline expert will be able to leave all of that to the computers in the back room – he or she won't even have to sit in on the initial planning sessions thrashing out just how the problem is going to be solved. This gives the discipline expert more time to do what he or she is actually paid to do: improve his or her discipline. While the back-room machines tirelessly turn the crank, the discipline expert can be developing better analysis modules, embedding more knowledge into revised PIA wrappers as to how to exploit this code, and the like. The only time the discipline expert's attention will be diverted into the mundane task of actually turning the crank is when the wrapper phones home for advice on how to proceed in a case outside its built-in experience.

16 Slide pst_vgrf_0136 – Autonomous Solution System Near-Term Demonstration

The next serious research goal to be faced is to demonstrate that an autonomous solution system is possible. While the algorithm, itself, is expected to be reasonably simple, developing a "sufficiently rich environment" in which that algorith can operate could be a major effort. The nature of the "sufficiently rich environment" is that it actual have a complete solution of the posed problem in it: applications that produce designs, analyses that consume designs and produce characteristics, and further analyses that consume characteristics and produce assessments of the merit of the design. Such an environment for real problems may only be expected after years of application of PIA to a problem area.

Clearly, a much more confined problem space is needed to demonstrate the possibility of an autonomous solution system algorithm. The area of simple mathematical theorems seems to offer the most modest possibilities. In particular, it is thought that a "sufficiently rich environment" for the autonomous solution of a quadratic equation might be within reach. That is, an envirnoment in which a posed quadratic equation of the form

$$ax^2 + bx + c = 0$$

could be solved to produce the quadratic formula,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In particular, an environment containing only the axioms

$$exp_a + c = exp_b \;\Rightarrow\; exp_a = exp_b - c$$

$$ax^2 + bx = c \;\Rightarrow\; 2ax + b = \sqrt{4ac + b^2}$$

$$exp_a x = exp_b \;\Rightarrow\; x = \frac{exp_b}{exp_a}$$

should be sufficient for an algorithm to develop the quadratic formula.

Having applied the various discovered axioms to develop the quadratic formula, the algorithm would then publish that result as a new theorem available within the PIA environment. The demonstration would be completed by re-posing the quadratic equation problem and observing the algorithm to proceed directly to its solution by applying the newly-publish quadratic formula theorem, rather than by re-developing that theorem by the application of the published axioms.

There is a key difference between the algorithm proposed for this problem and one for the autonomous solution of engineering problems: this algorithm works in the opposite direction. The proposed engineering algorithm works backward from a desired result while the mathematical algorithm works forward from a stated problem. Be this as it may, the key element of the demonstration is that both algorithms can, on an autonomous basis, search the PIA environment looking for applications that perform needed functions and produce needed results, arrange and apply those applications to reach a goal, and then record the results of that action for later consideration as the need may arise.

At first blush, this demonstration may not seem much of an advance over the accomplishments of the existing field known as Automatic Theorem Provers (ATPs); however, there is a small difference. ATPs, in general, must be supplied with a "conjecture"; that is, a statement of their goal or end state. The ATP then performs known symbolic manipulations to see if it can get from A to B. Thus, in our case, the ATP would have to be told that its invoker believes that, given a quadratic equation as a starting point, the quadratic formula might be true. But in PIA's formulation of this algorith, it is not told an end point; that is, it is not told that the quadratic formula might be its result. Instead, PIA is merely asked to look around at its resources and see where it can get to with a quadratic equation as its starting point. The algorithm not only can find and apply resources, but it can know that when it reaches an equation of the form

$$x = \{expression\ independent\ of\ x\}$$

it has reached a good end point.

Beyond this small advance lies another: in general, ATPs do not automatically incorporate the results of their activities so as to ease their task should they encounter the same problem, possibly as a component of a larger problem, again. In effect, ATPs do not learn and grow smarter. On the other

hand, PIA can encapsulate the result of its operation, publish it, recognize it, and apply it at a later time should that prove to be appropriate.

17 Slide pst_vgrf_0135 – CORBA Migration Benefits

PIA goals embraced from the beginning large, integrated analysis efforts. As such it was assumed without question that the final implementation would be in a net-enabled form that allowed many computers to partcipate in (and devote their energy to) a coordinated effort. From that standpoint alone, migration to a standard such as the Common Object Request Broker Architecture (CORBA) was a given.

One of the aspects of bringing many computers into a cooperating PIA environment was the nearly-limitless expansion of resources. Beyond the simple, raw expansion in compute power, a necessary expansion in storage space also accrues from the re-implementation in CORBA technology. One of the things learned in the C++ prototyping phase is that PIA creates objects (and, regrettably, overhead) at a significant rate, a very significant rate. Unfortunate as this additional overhead may be, it must be remembered that PIA is enabling capabilities never before realized such as the ability to produce a ful, auditable record of the development process.

The CORBA re-implementation provides a means, though, to at least pay this additional expense. CORBA allows objects to be deactivated (technically, *etherealized*); when a task arrises for a deactivated object, CORBA provides a mechanism to re-activate that object (technically, it is *incarnated*) and perform the necessary work. PIA uses these basic mechanisms to create, in effect, a second kind of virtual address space: object activity is tracked and objects that don't seem to be doing much are *etherialized*, their internal state being stored on secondary storage. When (and if) a method delivery does come in for an *etherealized* object, PIA uses the CORBA mechanisms to re-create that object, restore its internal state, and return the object to active operation.

The net effect of this use of object etherealization and incarnation is to extend the PIA data storage space to a practical infinity. Objects (or, more exactly, their internal state files) may be stored to the limit of the capacity of all the disk drives that may be attached to all the computers that may be joined as servers to any PIA collective. The PIA "name space" allows this set to extend to all the computers that can be joined to the Internet. Theoretically, all the computers of the entire world could be applied to hosting and storing information through PIA.

The CORBA implementation of PIA includes the concept of a server cluster: that is, a group of two or more hosts each serving an identical set of interfaces and each capable of serving any particular object created by any other member of the cluster. If the creating host is not available at the time a method is delivered, any other host of a particular collective is still able to serve the identified object. This significantly enhances reliability and availability of PIA information.

A PIA server cluster need not be physically co-located: a cluster may be formed with local and remote machines as long as they have access to the same persistent storage capabilities. This allows the effect of "hot-siting" to be created, allowing PIA information to be served even though an entire site might become unavailable.

The CORBA implementation of PIA also includes greater flexibility in persistent storage options. When the state of an object is to be saved, options are implemented to allow that state to be saved in more than one place. For example, PIA can be configured to save objects states in redundant local storage servers and redundant remote-site storage servers. PIA also anticipates the case in which inter-corporate concerns dictate the saving of object states not on server resources, but on the storage servers of a client instead.

Another feature of the CORBA re-implementation of PIA is that PIA now becomes implicitly a multi-user system. CORBA provides no base mechanism for restricting access to a particular user or client machine. Any user or client that can identify a CORBA-served object can deliver method invokations to it.

With the ability for multiple accessors provided, PIA then carries forward into the concept of multiple processes for a single user. In the comprehensive analysis of multiple configurations of a complex system there is implicit parallelism: sibling configurations of a system are in fact independent of each other, as are sibling analyses of each particular configuration. PIA recognizes these implicit parallelisms and, as an option, will spawn multiple processes to carry these independent tasks out. Thus, a single "user" may have PIA carrying out hundreds and thousands of processes at a time in the performance of a complex system analysis.

Also, CORBA holds out the promise of cross-language access to information. In theory, a JAVA client could access a C++ server of information. Actual practice of this theory has been, predictably, a little more difficult than the theory.

A final benefit of the CORBA migration of PIA technology is that it enables a new mode of software delivery. The functionality of an application can be served through a PIA-compliant, CORBA-served wrapper by a developer of a software application without the necessity of releasing the actual application code to the consumer. In addition to securing the code against piracy, the potential revelation of proprietary techniques, and the like, this technology may also significantly reduce software maintenance costs since the developer need only maintain the software copies actually resident on his own servers. Production of current-release media, shipping media to customers, and installation of revised software on customer machines may all be eliminated. While this does eliminate what may well be a profitable business unit for the provider of software, it correspondingly eliminates a bothersome overhead for the customer and allows him to devote more of his resources to productive effort. Correspondingly, this allows the developer to devote more of his capital resources to product development and maintenance of the product's competitive position in the market, rather than simply providing office space, production facilities, and the like, for a software maintenance unit.